Proof-Number Search and its Variants

H. Jaap van den Herik and Mark H.M. Winands

MICC-IKAT, Maastricht University, Maastricht, The Netherlands
{herik,m.winands}@micc.unimaas.nl

Summary. Proof-Number (PN) search is a best-first adversarial search algorithm especially suited for finding the game-theoretical value in game trees. The strategy of the algorithm may be described as developing the tree into the direction where the opposition characterised by value and branching factor is to expect to be the weakest. In this chapter we start by providing a short description of the original PN-search method, and two main successors of the original PN search, i.e., PN^2 search and the depth-first variant *Proof-number and Disproof-number Search* (PDS). A comparison of the performance between PN, PN^2, PDS, and $\alpha\beta$ is given. It is shown that PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in the game of Lines of Action (LOA). However, memory problems make the plain PN search a weaker solver for harder problems. PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. But PN^2 is restricted by its working memory, and PDS is considerably slower than PN^2. Next, we present a new proof-number search algorithm, called PDS-PN. It is a two-level search (like PN^2), which performs at the first level a depth-first PDS, and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS. Experiments show that within an acceptable time frame PDS-PN is more effective for really hard endgame positions. Finally, we discuss the depth-first variant df-pn. As a follow up of the comparison of the four PN variants, we compare the algorithms PDS and df-pn. However, the hardware conditions of the comparison were different. Yet, experimental results provide promising prospects for df-pn. We conclude the article by seven observations, three conclusions, and four suggestions for future research.

1 Endgame Solvers

Most modern game-playing computer programs use the adversarial search method called the $\alpha\beta$ algorithm [16] for online game-playing [11]. However, the $\alpha\beta$ search even with its enhancements is sometimes not sufficient to play well in the endgame. A variety of factors may cause this lack of effectiveness, for instance the complexity (in Go) and the depth of search (in many endgames). In some games, such as Chess, the latter problem is solved by the use of endgame databases [22]. Due to memory constraints this solution

is only feasible for endgames with a relatively small state-space complexity, although nowadays the size may be considerable.

In the last three decades many other search approaches have been proposed, tested and thoroughly investigated (for an overview see [12]). Two lines of research focused on the possibilities of the opponent and the potential threats of the opponent. The development started with the idea of conspiracy-number search as developed by McAllester [17] and worked out by Schaeffer [30]. This idea was heuristical by nature. It inspired Allis [1] to propose PN Search, a specialised binary (win or non-win) search method for solving games and for solving difficult endgame positions [2].

PN search is a best-first adversarial search algorithm especially suited for finding the game-theoretical value in game trees. In many domains PN search outperforms $\alpha\beta$ search in proving the game-theoretic value of endgame positions. The PN-search idea is a heuristic, which prefers expanding slim subtrees over wide ones. PN search or a variant thereof has been successfully applied to the endgame of Awari [2], Chess [6], Checkers [31, 32, 33], Shogi [34], and Go [15]. Since PN search is a best-first search, it has to store the whole search tree in memory. When the memory is full, the search has to end prematurely.

To overcome this problem PN^2 was proposed by Allis [1] as an algorithm to reduce memory requirements in PN search. It is elaborated upon in Breuker [5]. Its implementation and testing for chess positions is extensively described in Breuker, Uiterwijk, and Van den Herik [8]. PN^2 performs two levels of PN search, one at the root and one at the leaves of the first level. As in the B* algorithm [4], a search process is started at the leaves to obtain a more accurate evaluation. Although PN^2 uses far less memory than PN search, it does not fully overcome the memory obstacle.

Therefore, the idea behind the MTD(f) algorithm [25] was applied to PN variants: try to construct a depth-first algorithm that behaves as its corresponding best-first search algorithm. This idea became a success. In 1995, Seo formulated a depth-first iterative-deepening version of PN search, later called PN* [34]. The advantage of this variant is that there is no need to store the whole tree in memory. The disadvantage is that PN* is slower than PN [29].

Other depth-first variants are PDS [18] and df-pn [21]. Although their generation of nodes is even slower than PN*'s, they are building smaller search trees. Hence, they are in general more efficient than PN*.

In this chapter we will investigate several PN-search algorithms, using the game of *Lines of Action (LOA)* [26] as test domain. We will concentrate on the *offline* application of the PN-search algorithms. The number of positions they can solve (i.e., the post-mortem analysis quality) is tested on a set of endgame positions. Moreover, we will investigate to what extent the algorithms are restricted by their working memory *or* by the search speed.

The chapter is organised as follows. In Section 1 we discuss the need for special algorithms to solve endgame positions. Section 2 describes PN, PN^2, PN*, PDS, and df-pn. In Section 3 two enhancements of PN and PN^2 are

described. In Section 4 we examine the offline solution power and the solution time of three PN variants, in relation to those of $\alpha\beta$. In Section 5 we explain the working of PDS-PN by elaborating on PDS and the idea of two-level search algorithms. Then, in Section 6, the results of experiments with PDS-PN on a set of endgame positions are given. Subsequently, we briefly discuss df-pn and compare the results by PDS and df-pn in Section 7. Finally, in Section 8 we present seven observations, three conclusions and four suggestions for future research.

2 Five Proof-Number Search Algorithms

In this section we give a short description of PN search (Subsection 2.1), PN^2 search (Subsection 2.2) and three depth-first variants of PN search . Recently, three depth-first PN variants, PN*, PDS, and df-pn have been proposed, which solved the memory problem of PN-search algorithms. They will be discussed in Subsection 2.3, 2.4, and 2.5.

2.1 Proof-Number Search

Proof-Number (PN) search is a best-first search algorithm especially suited for finding the game-theoretical value in game trees [1]. Its aim is to prove the true value of the root of a tree. A tree can have three values: *true*, *false*, or *unknown*. In the case of a forced win, the tree is *proved* and its value is true. In the case of a forced loss or draw, the tree is *disproved* and its value is false. Otherwise the value of the tree is unknown. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function to determine the most-promising node to be expanded next [2]. In PN search this node is usually called the *most-proving* node. PN search selects the most-proving node using two criteria: (1) the shape of the search tree (the branching factor of every internal node) and (2) the values of the leaves. These two criteria enable PN search to treat game trees with a non-uniform branching factor efficiently. The strategy of the algorithm may be described as developing the tree into the direction where the opposition characterised by value and branching factor is to expect to be the weakest.

Below we explain PN search on the basis of the AND/OR tree depicted in Figure 1, in which a square denotes an OR node, and a circle denotes an AND node. The numbers to the right of a node denote the proof number (upper) and disproof number (lower). A *proof number* (*pn*) represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a *disproof number* (*dpn*) represents the minimum number of leaf nodes which have to be disproved in order to disprove the node. Because the goal of the tree is to prove a forced win, winning nodes are regarded as proved. So, they have proof number 0 and disproof number ∞ (e.g., node i). Lost or drawn nodes are regarded as disproved (e.g., nodes f and k). They have proof

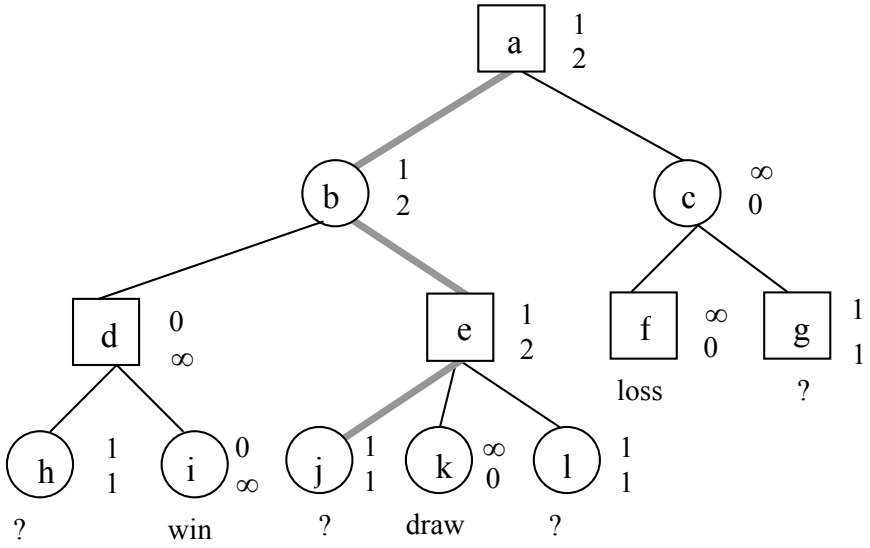


Fig. 1. An AND/OR tree with proof and disproof numbers

number ∞ and disproof number 0. Unknown leaf nodes have a proof and disproof number of unity (e.g., nodes *g*, *h*, *j*, and *l*). The proof number of an internal AND node is equal to the sum of its children's proof numbers, since to prove an AND node all the children have to be proved. The disproof number of an AND node is equal to the minimum of its children's disproof numbers, since to disprove an AND node it suffices to disprove one child. The proof number of an internal OR node is equal to the minimum of its children's proof numbers, since to prove an OR node it suffices to prove one child. The disproof number of an internal OR node is equal to the sum of its children's disproof numbers, since to disprove an OR node all the children have to be disproved.

The procedure of selecting the most-proving node to expand is as follows. We start at the root. Then, at each OR node the child with the lowest proof number is selected as successor, and at each AND node the child with the lowest disproof number is selected as successor. Finally, when a leaf node is reached, it is expanded (which makes the leaf node an internal node) and the newborn children are evaluated. This is called *immediate evaluation*. The selection of the most-proving node (*j*) in Figure 1 is given by the bold path.

The number of node traversals to select the most-proving node can have a negative impact on the execution time. Therefore, Allis [1] proposed the following minor enhancement. The updating process can be terminated when the proof and disproof number of a node do not change. From this node we can start the next most-proving node selection. For an adequate description of implementation details we refer to Allis *et al.* [2], where the essentials for implementation are given.

In the naive implementation, proof and disproof numbers are each initialised to unity in the unknown leaves. In other implementations, the proof number and disproof number are set to 1 and n, respectively, for an OR node (and the reverse for an AND node), where n is the number of legal moves. In LOA this would mean that we take the *mobility* of the moving player in the position into account, which is an important feature in the evaluation function as well [37]. The effect of this enhancement is tested in Section 3. We would like to remark that there are other possibilities to initialise the proof and disproof numbers. Allis [1] applies domain-specific knowledge to set the variables in Awari. Saito *et al.* [27] apply an algorithm called MC-PNS in the game of Go. It gives a value to the proof and disproof number by performing a Monte-Carlo evaluation at the leaf node of the tree.

Here we reiterate that a disadvantage of PN search is that the whole search tree has to be stored in memory. When the memory is full, the search process has to be terminated prematurely. A partial solution is to delete proved or disproved subtrees [1]. In the next subsections we discuss two main variants of PN search that handle the memory problem more adequately.

2.2 PN^2 Search

For an appropriate description we repeat a few sentences of our own. PN^2 is first described by Allis [1], as an algorithm to reduce memory requirements in PN search. It is elaborated upon in Breuker [5]. Its implementation and testing for chess positions is extensively described in Breuker *et al.* [8]. PN^2 consists of two levels of PN search. The first level consists of a PN search (PN_1), which calls a PN search at the second level (PN_2) for an evaluation of the most-proving node of the PN$_1$-search tree. This PN$_2$ search is bound by a maximum number of nodes to be stored in memory. The number is a fraction of the size of the PN$_1$-search tree. The fraction *f(x)* is given by the logistic-growth function [3], x being the size of the first-level search:

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \quad , \tag{1}$$

with parameters a and b, both strictly positive. The number of nodes y in a PN$_2$-search tree is restricted to the minimum of this fraction function and the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x), \tag{2}$$

with N the maximum number of nodes to be stored in memory.

The PN$_2$ search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proved. After completion of the PN$_2$ search, the children of the root of the PN$_2$-search tree are preserved, but subtrees are removed from memory. The children of the most-proving node (the root of the PN$_2$-search tree) are not immediately evaluated by a second-level search;

evaluation of such a child node happens only after its selection as most-proving node. This is called *delayed evaluation*. We remark that for PN_2-search trees immediate evaluation is used. The essentials of our implementation are given in [5].

As we have seen in Subsection 2.1, proved or disproved subtrees can be deleted. If we do not delete proved or disproved subtrees in the PN_2 search the number of nodes searched is the same as y, otherwise we can continue the search longer. The effect of *deleting (dis)proved PN_2 subtrees* is tested in Section 3.

2.3 PN*

In 1995, Seo formulated the first depth-first iterative-deepening version of PN search, later called PN* [34]. PN* uses a method called *multiple-iterative deepening*. Instead of iterating only at the root node such as in the ordinary iterative deepening, it iterates also at AND nodes. To each AND node a threshold is given. The subtree rooted at that node is continued to be searched as long as the proof number is below the assigned threshold. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a transposition table.

2.4 PDS

The disadvantage of PN* is that it has difficulties to disprove a (sub)tree, which harms its solving performance [29]. Nagai [18, 19] proposed a second depth-first search algorithm, called *Proof-number and Disproof-number Search* (PDS), which is a straight extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too.[1] Moreover, PDS uses *multiple-iterative deepening* in *every* node. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a *TwoBig* transposition table [7]. PDS uses two thresholds in searching, one for the proof numbers and one for the disproof numbers. We note that PDS suffers from the Graph-History Interaction (GHI) problem (cf. [9]).[2] In the present implementation this problem is ignored [19]. In Section 5.2 we will describe PDS in detail.

2.5 df-pn

Nagai [20, 21] has introduced a third depth-first PN algorithm, called df-pn (depth-first proof-number search). It is mainly a variant of PDS. The

[1] We recall that PN and PN^2 use disproof numbers too.

[2] In a search graph a node's value may be dependent on the path leading to it. Different paths may lead to different values. Hence, it is difficult to determine the value of any node unambiguously. The problem is known as the Graph-History Interaction (GHI) problem (see [10, 23]).

algorithm df-pn does not perform iterative deepening at the root node. As with PDS, df-pn uses two thresholds for a node, one as a limit for proof numbers (pnt) and one for disproof numbers (dnt). In contrast to PDS, it sets the thresholds of both proof number and disproof number at the root node to ∞. Once the thresholds are assigned to a node, the subtree rooted in that node is stopped to be searched if the proof number (pn) or disproof number (dpn) is larger than or equal to their corresponding threshold. Obviously, the condition $pn < \infty$ and $dn < \infty$ holds if the tree is not solved. As the search goes more deeply, the threshold values are distributed among the descendant nodes.

At an OR node p we select the child n with lowest pn just like in the regular PN search. Assume that there is a node s with the second lowest pn value. The thresholds of node n are set in the following way:

$$pnt_n = min(pnt_p, pn_s + 1), \tag{3}$$
$$dnt_n = dnt_p - dn_p + dn_n. \tag{4}$$

Similarly, at an AND node p we select the child n with lowest dn just like in the regular PN search. Assume that there is a node s with the second lowest dpn value. The thresholds of node n are set in the following way:

$$pnt_n = pnt_p - pn_p + pn_n, \tag{5}$$
$$dnt_n = min(dnt_p, dpn_s + 1). \tag{6}$$

Contrary to PDS, it has been proved that df-pn always selects the most-proving node [20]. Initially, the results of df-pn were mixed [21, 28]. Although df-pn sometimes solves positions faster than PDS, it may solve in practice fewer positions [28]. It turns out that the standard df-pn algorithm suffers more from the GHI problem than PDS. It has a fundamental problem when applied to a domain with repetitions [14]. Nagai proposed an ad-hoc way to solve this problem for Tsume-Shogi [20]. Recently, Kishimoto and Müller proposed a solution, which adequately handles the GHI problem in df-pn [13, 15].

To prevent multiple recreations of a subtree due to multiple-iterative deepening, Pawlewicz and Lew [24] developed the $1 + \epsilon$ *trick* enhancement that might improve df-pn considerably. It transforms formula 3 to formula 7 for the child's proof-number threshold in a OR node:

$$pnt_n = min(pnt_p, \lceil pn_s(1 + \epsilon) \rceil), \tag{7}$$

where ϵ is a small real number greater than zero.

3 Two Enhancements of PN and PN^2

Below, we test the effect of enhancing PN and PN^2 with (1) adding mobility and (2) deleting (dis)proved PN_2 subtrees. For PN and PN^2, all nodes evaluated for the termination condition during the search are counted. The node count is equal to the number of nodes generated. The maximum number of nodes searched is 50,000,000. The maximum number of nodes stored in memory is 1,000,000. These numbers roughly corresponds to the tournament conditions of the Computer Olympiad with respect to Pentium III. The parameters (a,b) of the growth function used in PN^2 are set at (1800K, 240K) according to the suggestions in Breuker *et al.* [8].

In the first experiment, we tested PN search and PN^2 with the mobility enhancements on a test set of 116 LOA positions.[3] The results are shown in Table 1. In the second column we see that PN search solved 85 positions using mobility; without mobility it solved 53 positions. PN^2 search using mobility solved 109 positions and without it solved only 91 positions. Next, in the third column we see that on a set of 53 positions solved by both PN algorithms, PN search using mobility is roughly 5 times faster in nodes than PN search without mobility. Finally, in the fourth column we see that on a set of 91 positions solved by both PN^2 algorithms, PN^2 search using mobility is more than 6 times faster in nodes than PN^2 search without using mobility. In general we may conclude that mobility speeds up the PN and PN^2 with a factor 5 to 6. The time spent on the mobility extension is estimated at 15% of the total computing time. Owing to mobility PN search can make better search decisions and therefore solve many more positions. The underlying reason is that the memory constraint is violated less frequently.

Table 1. Mobility in PN and PN^2

Algorithm	# of pos. solved	Total nodes (53 pos.)	Total nodes (91 pos.)
PN	53	24,357,832	-
PN + Mob.	85	5,053,630	-
PN^2	91	-	345,986,639
PN^2 + Mob.	109	-	56,809,635

In the second experiment, we tested the effect of deleting (dis)proved subtrees at the PN_2 search of the PN^2. The results are shown in Table 2. Both variants (not deleting PN_2 subtrees and deleting PN_2 subtrees) used mobility in the experiment. On a set of 108 positions that both versions were able to solve, we can see that deleting (dis)proved subtrees improves the search by 10%. It also solves one additional position.

In the remainder of this chapter we will use these two enhancements (i.e., mobility and deleting (dis)proved PN_2 subtrees) for PN and PN^2.

[3] The test set can be found at www.cs.unimaas.nl/m.winands/loa/tswin116.zip.

Table 2. Deleting (dis)proved subtrees at the second-level search PN^2

Algorithm	# of pos. solved	Total nodes (108 pos.)
PN^2 not deleting PN_2 subtrees	108	463,076,682
PN^2 deleting PN_2 subtrees	109	416,168,419

4 PN Search Performance

In this section we test the offline performance of three PN-search variants by comparing PN, PN^2, PDS, and $\alpha\beta$ search with each other. The goal is to investigate the effectiveness of the PN-search variants by experiments. We will look how many endgame positions they can solve and how much effort (in nodes and CPU time) they take. For the $\alpha\beta$ depth-first iterative-deepening search, nodes at depth i are counted only during the first iteration that the level is reached. This is how analogous comparisons are done in Allis [1]. For PN, PN^2, and PDS search, all nodes evaluated for the termination condition during the search are counted. For PDS this node count is equal to the number of expanded nodes (function calls of the recursive PDS algorithm). PN, PN^2, PDS, and $\alpha\beta$ are tested on a set of 488 forced-win LOA positions.[4] Two comparisons are made, which are described in Subsection 4.1 and 4.2.

4.1 A General Comparison of Four Search Techniques

In Table 3 we compare PN, PN^2, PDS, and $\alpha\beta$ on a set of 488 LOA positions. The maximum number of nodes searched is again 50,000,000. In the second column of Table 3 we see that 470 positions were solved by the PN^2 search, 473 positions by PDS, only 356 positions by PN, and 383 positions by $\alpha\beta$. In the third and fourth column the number of nodes and the time consumed are given for the subset of 314 positions, which all four algorithms were able to solve. If we have a look at the third column, we see that PN search builds the smallest search trees and $\alpha\beta$ by far the largest. PN^2 and PDS build larger trees than PN but can solve significantly more positions. This suggests that both algorithms are better suited for harder problems. PN^2 investigates 1.2 times more nodes than PDS, but PN^2 is (more than) 6 times faster than PDS in CPU time for this subset.

From the experiments we may draw the following three conclusions.

1. PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in LOA.
2. The memory problems make the plain PN search a weaker solver for the harder problems.
3. PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$.

[4] The test set can be found at www.cs.unimaas.nl/m.winands/loa/tscg2002a.zip.

Table 3. Comparing the search algorithms on 488 test positions

Algorithm	# of positions solved (out of 488)	314 positions	
		Total nodes	Total time (ms.)
$\alpha\beta$	383	1,711,578,143	22,172,320
PN	356	89,863,783	830,367
PN^2	470	139,254,823	1,117,707
PDS	473	118,316,534	6,937,581

4.2 A Deep Comparison of PN^2 and PDS

For a better insight into how much faster PN^2 is than PDS in CPU time, we did a second comparison. In Table 4 we compare PN^2 and PDS on the subset of 463 test positions, which both algorithms were able to solve. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the decrease of performance is that for hard problems the PN_2-search tree becomes as large as the PN_1-search tree. Therefore, the PN_2-search tree is causing more overhead. However, if we have a look at the CPU times we see that PN^2 is still three times faster than PDS. The reason is that PDS has a relatively large time overhead because of the delayed evaluation. Consequently, the number of nodes generated is higher than the number of nodes expanded. In our experiments, we observed that PDS generated nodes 7 to 8 times slower than PN^2. Such a figure for the overhead is in agreement with experiments performed in Othello and Tsume-Shogi [29]. We remark that Nagai's [19] Othello results showed that PDS was better than PN search, (i.e., it solved the positions faster than PN). Nagai assigned to both the proof number and the disproof number of unknown nodes a 1 in his PN search and therefore did not use the mobility enhancement. In contrast, we incorporated the mobility in the initialisation of the proof numbers and disproof numbers in our PN search. We believe that comparing PDS with a PN-search algorithm without using the mobility component is not fair. Since PDS does not store unexpanded nodes that have a proof number 1 and disproof number 1, we may state that PDS initialises the (dis)proof number of a node by counting the number of its newborn children [19]. So in the PDS search, the mobility enhancement coincides with the initialisation of the (dis)proof number.

From this second experiment we may conclude that PDS is considerably slower than PN^2 in CPU time. Therefore, PN^2 seems to be a better endgame solver under tournament conditions. Counterbalancing this success, we note that PN^2 is still restricted by its working memory and is not fit for solving really hard problems.

Table 4. Comparing PDS and PN^2 on 463 test positions

Algorithm	Total nodes	Total time (ms.)
PN^2	1,462,026,073	11,387,661
PDS	562,436,874	34,379,131

5 PDS-PN

In Section 4 we have observed two facts: (1) the advantage of PN^2 over PDS is that it is faster and (2) the advantage of PDS over PN^2 is that its tree is constructed as a depth-first tree, which is not restricted by the available working memory. In the next sections we try to overcome fact (1) while preserving fact (2) by presenting a new proof-number search algorithm, called PDS-PN [35, 36]. It is a two-level search (like PN^2), which performs at the first level a depth-first Proof-number and Disproof-number Search (PDS), and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS. In this section we give a description of PDS-PN search, which is a two-level search using PDS at the first level and PN at the second level. In Subsection 5.1 we motivate why we developed the method. In Subsection 5.2 we describe the first-level PDS, and in Subsection 5.3 we provide background information on the second-level technique. Finally, in Subsection 5.4 the relevant parts of the pseudo code are given.

5.1 Motivation

The development of the PDS-PN algorithm was motivated by the clear advantage that PDS is traversing a depth-first tree instead of a best-first tree. Hence, PDS is not restricted by the available working memory. As against this, PN has the advantage of being fast compared to PDS (see Section 4).

The PDS-PN algorithm is designed to combine the two advantages. At the first level, the search is a depth-first search, which implies that PDS-PN is not restricted by memory. At the second level the focus is on fast PN. It is a complex balance, but we expect that PDS-PN will be faster than PDS, and PDS-PN will not be hampered by memory restrictions. Since the expectation on the effectiveness of PDS-PN is difficult to prove we have to rely on experiments (see Section 6). In the next two subsections we describe PDS-PN.

5.2 First Level: PDS

PDS-PN is a two-level search like PN^2. At the first level a PDS search is performed, denoted PN_1. For the expansion of a PN_1 leaf node, not stored in the transposition table, a PN search is started, denoted PN_2.

Proof-number and Disproof-number Search (PDS) [18] is a straightforward extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too. PDS exploits a method called *multiple-iterative deepening*. Instead of iterating only in the root such as in ordinary iterative deepening, PDS iterates in *all* interior nodes. The advantage of using the multiple-iterative-deepening method is that in most cases it accomplishes to select the most-proving node (see below), not only in the root, but also in the interior nodes of the search tree. To keep iterative deepening effective, the

method is enhanced by storing the expanded nodes in a *TwoBig* transposition table [7].

PDS uses two thresholds for a node, one as a limit for proof numbers and one for disproof numbers. Once the thresholds are assigned to a node, the subtree rooted in that node is stopped to be searched if both the proof number and disproof number are larger than or equal to the thresholds *or* if the node is proved or disproved. The thresholds are set in the following way. At the start of every iteration, the proof-number threshold *pnt* and disproof-number threshold *dnt* of a node are equal to the node's proof number *pn* and disproof number *dn*, respectively. If it seems more likely that the node can be proved than disproved (called *proof-like*), the proof-number threshold is increased. If it seems more likely that the node can be disproved than proved (called *disproof-like*), the disproof-number threshold is increased. In passing we note that it is easier to prove a tree in an OR node, and to disprove a tree in an AND node. Below we repeat Nagai's [18] heuristic to determine proof-like and disproof-like.

In an interior OR node n with parent p (direct ancestor) the solution of n is proof-like, if the following condition holds:

$$pnt_p > pn_p \;\; AND \;\; (pn_n \leq dn_n \;\; OR \;\; dnt_p \leq dn_p), \tag{8}$$

otherwise, the solution of n is disproof-like.

In an interior AND node n with parent p (direct ancestor) the solution of n is disproof-like, if the following condition holds:

$$dnt_p > dn_p \;\; AND \;\; (dn_n \leq pn_n \;\; OR \;\; pnt_p \leq pn_p), \tag{9}$$

otherwise, the solution of n is proof-like.

When PDS does not prove or disprove the root given the thresholds, it increases the proof-number threshold if its proof number is smaller than or equal to its disproof number, otherwise it increases the disproof-number threshold. Finally, we remark that only expanded nodes are evaluated. This is called *delayed evaluation* (cf. [1]). The expanded nodes are stored in a transposition table. The proof and disproof number of a node are set to unity when not found in the transposition table. Since PDS does not store unexpanded nodes which have a proof number 1 and disproof number 1, it can be said that PDS initialises the proof and disproof number by using the number of children. The mobility enhancement of PN and PN^2 (see Subsection 2.1) is already implicitly incorporated in the PDS search.

PDS is a depth-first search algorithm but behaves like a best-first search algorithm. In most cases PDS selects the same node for expansion as PN search. By using transposition tables PDS suffers from the GHI problem (cf. [9]). Especially the GHI evaluation problem can occur in LOA too. For instance, draws can be agreed upon due to the three-fold-repetition rule. Thus, dependent on its history a node can be a draw or can have a different value.

However, in the current PDS algorithm we ignore this problem, since we believe that it is less relevant for the game of LOA than for Chess.

A Detailed Example

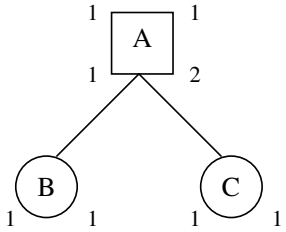
A detailed step-by-step example of the working of PDS is given in Figure 2. A square denotes an OR node, and a circle denotes an AND node. The numbers at the upper side of a node denote the proof-number threshold (left) and disproof-number threshold (right). The numbers at the lower side of a node denote the proof number (left) and disproof number (right).

In the first iteration (top of Figure 2), threshold values of the root A are set to unity. A is expanded, and nodes B and C are generated. The proof number of A becomes 1 and the disproof number becomes 2. Because both numbers are larger than or equal to the threshold values the search stops.

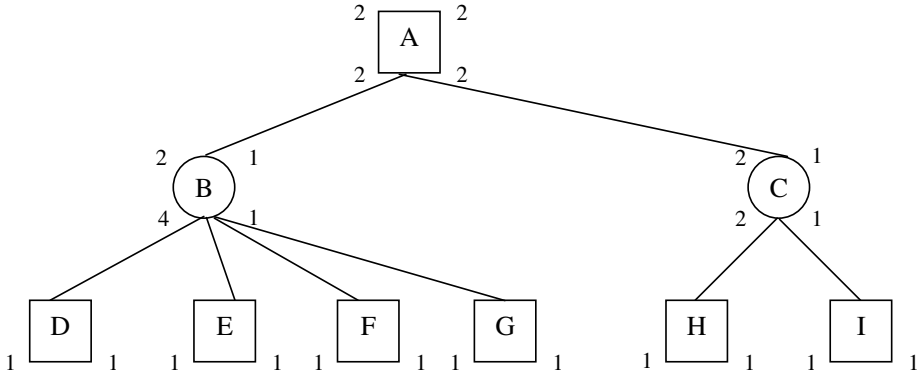
In the second iteration (middle of Figure 2), the proof-number threshold is incremented to 2, because the proof number of A (i.e., 1) is the smaller one of both A's proof number and disproof number (i.e., 2). We again expand A and re-generate B and C. The proof number of A is below its proof-number threshold and we continue searching. Now we have to select the child with minimum proof number. Because B and C have the same proof number, the left-most node B is selected. Initially, we set the proof-number and disproof-number threshold of B to its proof and disproof number (both 1). Because B is an AND node we have to look whether the solution of B is disproof-like by checking the appropriate condition (i.e., formula 9). The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution of B is not disproof-like but proof-like. Thus, the proof-number threshold of B has to be incremented to 2. Next, node B is expanded and the nodes D, E, F and G are generated. The search in node B is stopped because its proof number (i.e., 4) and disproof number (i.e., 1) are larger than or equal to the thresholds (i.e., 2 and 1, respectively). Node B is stored in the transposition table with proof number 4 and disproof number 1. Then the search backtracks to A. There we have to check whether we still can continue searching A. Since the proof number of A is smaller than its threshold, we continue and subsequently we select C, because this node has now the minimum proof number. The thresholds are set in the same way as in node B. Node C has two children H and I. The search at node C is stopped because its proof number (i.e., 2) and disproof number (i.e., 1) are not below the thresholds. C is stored in the transposition table with proof number 2 and disproof number 1. The search backtracks to A and is stopped because its proof number (i.e., 2) and disproof number (i.e., 2) are larger than or equal to the thresholds. We remark that at this moment B and C are stored because they were expanded.

In the third iteration (bottom of Figure 2) the proof-number threshold of A is incremented to 3. Nodes B and C are again generated, but this time we can find their proof and disproof numbers in the transposition table. The node with smallest proof number is selected (C with proof number 2). Initially,

Iteration 1:



Iteration 2:



Iteration 3:

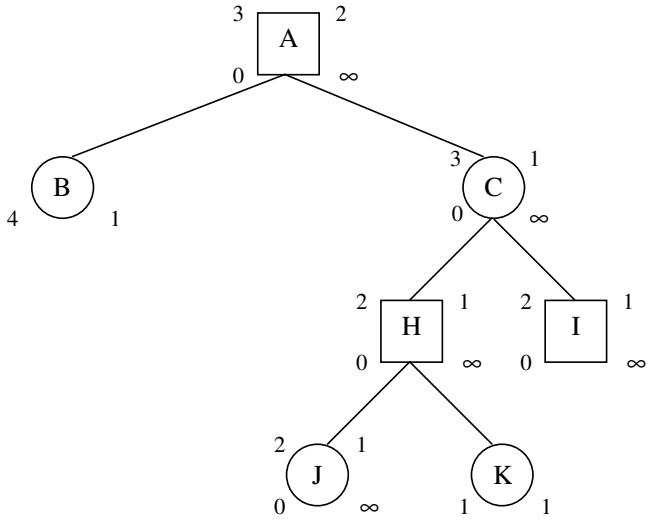


Fig. 2. An illustration of PDS

we set the proof-number threshold and disproof-number threshold of C to its proof and disproof number (i.e., 2 and 1, respectively). Because C is an AND node we have to look whether the solution is disproof-like by checking condition 9. The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution is not disproof-like but proof-like. Thus, the proof-number threshold of C has to be incremented to 3. C has now proof-number threshold 3 and disproof-number threshold 1. Nodes H and I are generated again by expanding C. This time the proof number of C (i.e., 2) is below the proof-number threshold (i.e., 3) and the search continues. The node with minimum disproof number is selected (i.e., H). Initially, we set the proof-number threshold and disproof-number threshold of H to its proof and disproof number (i.e., both 1). Because H is an OR node we have to look whether the solution is proof-like by checking condition 8. The proof-number threshold of C (i.e., 3) is larger than its proof number (i.e., 2), therefore the solution is proof-like. Hence, the search expands node H with proof-number threshold 2 and disproof-number threshold 1. Nodes J and K are generated. Because the proof number of H (i.e., 1) is below its threshold (i.e., 2), the node with minimum proof number is selected. Because J is an AND node we have to look whether the solution of J is disproof-like by checking condition 9. The disproof-number threshold of H (i.e., 1) is not larger than its disproof number (i.e., 2), therefore the solution of J is not disproof-like but proof-like. J is expanded with proof-number threshold 2 and disproof-number threshold 1. Since node J is a terminal win position its proof number is set to 0 and its disproof number set to ∞. The search backtracks to H. At node H the proof number becomes 0 and the disproof number ∞, which means the node is proved. The search backtracks to node C. The search continues because the proof number of C (i.e., 1) is not larger than or equal to the proof-number threshold (i.e., 3). We select now node I because it has the minimum disproof number. The thresholds of node I are set to 2 and 1, as was done in H. The node I is a terminal win position; therefore its proof number is set to 0 and its disproof number to ∞. At this moment the proof number of C is 0 and the disproof number ∞, which means that the node is proved. The search backtracks to A. The proof number of A becomes 0, which means that the node is proved. The search stops at node A and the tree is proved.

5.3 Second Level: PN Search

For an adequate description we reiterate a few sentences from Subsection 2.2. At the leaves of the first-level search tree, the second-level search is invoked, similar as in PN^2 search. The PN search of the second-level, denoted PN_2 search, is bounded by the number of nodes that may be stored in memory. The number is a fraction of the size of the PN_1-search tree, for which we take the current number of nodes stored in the transposition table of the PDS search. Preferably, this fraction should start small, and grow larger as the size of the first-level search tree increases. A standard model for this growth

is the logistic-growth model [3]. The fraction *f(x)* is therefore given by the logistic-growth function, x being the size of the first-level search:

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \ , \tag{10}$$

with parameters a and b, both strictly positive. The parameter a determines the transition point of the function: as soon as the size of the first-level search tree reaches a, the second-level search equals half the size of the first-level search. Parameter b determines the S-shape of the function: the larger b, the more stretched the S-shape is. The number of nodes y in a PN_2-search tree is restricted by the minimum of this fraction function and the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x), \tag{11}$$

with N the maximum number of nodes to be stored in memory.

The PN_2 search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proved. After completion of the PN_2-search tree, only the root of the PN_2-search tree is stored in the transposition table of the PDS search. We remark that for PN_2-search trees *immediate evaluation* (cf. [1]) is used. This two-level search is schematically sketched in Figure 3.

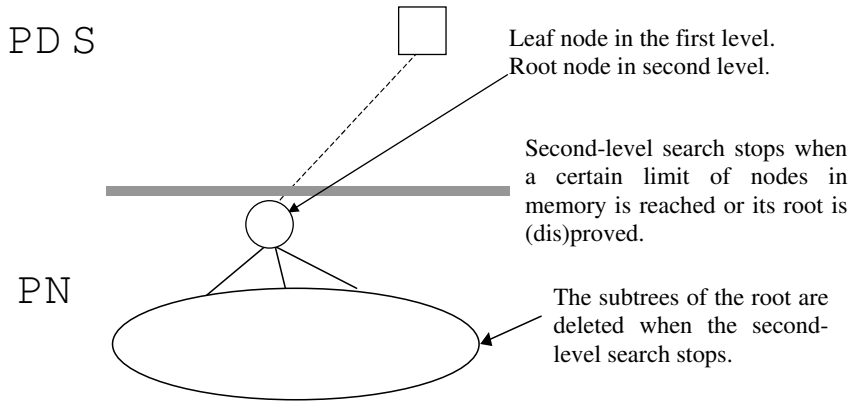


Fig. 3. Schematic sketch of PDS-PN

In the second-level search proved or disproved subtrees are deleted. If we do not delete proved or disproved subtrees in the PN_2 search, the number of nodes searched becomes the same as y. When we include deletions the second-level search can continue on average considerably longer.

```
//Iterative deepening at root r
procedure NegaPDSPN(r){

  r.proof = 1;
  r.disproof = 1;

  while(true){
    MID(r);
    //Terminate when the root is proved or disproved
    if(r.proof == 0 || r.disproof == 0)
      break;

    if(r.proof <= r.disproof)
      r.proof++;
    else
      r.disproof++;
  }
}
```

Fig. 4. PDS-PN: Root node

5.4 Pseudo Code for PDS-PN

In this subsection we provide the pseudo code for PDS-PN. For ease of comparison we use similar pseudo code as used by Nagai [18] for the PDS algorithm. The proof number in an OR node and the disproof number in an AND node are equivalent. Analogously, the disproof number in an OR node and the proof number in an AND node are equivalent. As they are dual to each other, an algorithm similar to negamax in the context of minimax searching can be constructed. This algorithm is called NegaPDSPN (see Figure 4).

In the following, procedure `MID(n)` performs multiple iterative deepening (see Figure 5). The function `proofSum(n)` computes the sum of the proof numbers of all the children. The function `disproofMin(n)` finds the minimum of the disproof numbers of all the children. The procedure `putInTT()` stores information to and `lookUpTT()` retrieves information from the transposition table. `isTerminal(n)` checks whether a node is a win, a loss, or a draw. The procedure `generateChildren(n)` generates the children of the node. By default, the proof number and disproof number of a node are set to unity. The procedure `findChildrenInTT(n)` checks whether the children are already stored in the transposition table. If a hit occurs for a child, its proof number and disproof number are set to the values found in the transposition table. The procedure `PN()` is just the plain PN search. The algorithm is described in Allis [1] and Breuker [5], and is reproduced in the Appendix . The function `computeMaxNodes()` computes the number of nodes, which may be stored for the PN search, according to Equation 11.

Finally, the function `selectChild()` selects the child that will be traversed next (see Figure 6).

```
procedure MID(n){
  //Look up in the transposition table
  lookUpTT(n, &proof, &disproof);
  if(proof == 0 || disproof == 0
  || (proof >= n.proof && disproof >= n.disproof)){
    n.proof = proof; n.disproof = disproof;
    return;
  }

  if(isTerminal(n)){
    if((n.value == true && n.type == AND_NODE)
    ||(n.value == false && n.type == OR_NODE)){
      n.proof = INFINITY; n.disproof = 0;
    }
    else{
      n.proof = 0; n.disproof = INFINITY;
    }
    putInTT(n);
    return;
  }

  generateChildren(n);
  //Avoid cycles
  putInTT(n);

  //Multiple-iterative deepening
  while(true){
    //Check whether the children are already stored in the TT.
     findChildrenInTT(n);
    //Terminate when both pn and dn exceed their thresholds
    if(proofSum(n) == 0 || disproofMin(n) == 0 || (n.proof <=
    disproofMin(n) && n.disproof <= proofSum(n))){
      n.proof = disproofMin(n);
      n.disproof = proofSum(n);
      putInTT(n);
      return;
    }

    proof = max(proof, disproofMin(n));
    n_child = selectChild(n, proof);

    if(n.disproof > proofSum(n) && (proof_child <= disproof_child
      || n.proof <= disproofMin(n)))
      n_child.proof++;
    else
      n_child.disproof++;

    //This is the PDS-PN part
     if(!lookUpTT(n_child)){
      PN(n_child, computeMaxNodes());
      putInTT(n_child);
    }
    else
      MID(n_child);
  }
}
```

Fig. 5. PDS-PN: Multiple-Iterative Deepening

```
//Select among children
selectChild(n, proof){

  min_proof = INFINITY;
  min_disproof  = INFINITY;
  for(each child n_child){
    disproof_child = n_child.disproof;
    if(disproof_child != 0)
      disproof_child = max(disproof_child, proof);

    //Select the child with the lowest disproof_child (if there are
    //plural children among them select the child with the lowest
    //n_child.proof)
    if(disproof_child < min_disproof || (disproof_child
       == min_disproof && n_child.proof < min_proof)){
      n_best = n_child;
      min_proof = n_child.proof;
      min_disproof = disproof_child;
    }
  }
  return n_best;
}
```

Fig. 6. PDS-PN: Selection mechanism

6 Experiments

In this section we compare $\alpha\beta$, PN^2, PDS, and PDS-PN search with each other. The goal is to prove experimentally the effectiveness of PDS-PN. We will investigate how many endgame positions it can solve and the effort (in terms of number of nodes and CPU time) it takes compared with $\alpha\beta$, PN^2, PDS. For PDS and PDS-PN we use a *TwoBig* transposition table. In Subsection 6.1 we test PDS-PN with different parameters a and b for the growth function. In Subsection 6.2 we compare PDS-PN with $\alpha\beta$, PN^2, and PDS on a set of 488 LOA positions in three different ways. In Subsection 6.3 we compare PDS-PN with PN^2 on a set of hard LOA problems. Finally, we evaluate the algorithms PDS-PN and PN^2 in solving problems under restricted memory conditions in Subsection 6.4.

6.1 Parameter Tuning

In the following series of experiments we measured the solving ability with different parameters a and b. For our specific parameter choice we follow Breuker [5], i.e., parameter a takes values of 150K, 450K, 750K, 1050K, and 1350K, and for each value of a parameter b takes values of 60K, 120K, 180K, 240K, 300K, and 360K. The results are given in Table 5. For each a holds that

the number of solved positions grows with increasing b, when the parameter b is still small. If b is sufficiently large, increasing it will not enlarge the number of solved positions. In the process of parameter tuning we found that PDS-PN solves the most positions with (450K, 300K) (see the bold line in Table 5). However, the difference with parameter configurations (150K, 180K), (150K, 240K), (150K, 300K), (150K, 360K), (450K, 360K), and (1350K, 300K) is not significant. On the basis of these results we decided that it is not necessary to perform experiments with a larger a.

Table 5. Number of solved positions (by PDS-PN) for different values of a and b

a	b	# of solved positions	accuracy (%)	a	b	# of solved positions	accuracy (%)
150,000	60,000	460	94.3	750,000	240,000	463	94.9
150,000	120,000	458	93.9	750,000	300,000	460	94.3
150,000	180,000	466	95.5	750,000	360,000	461	94.5
150,000	240,000	466	95.5	1,050,000	60,000	421	86.3
150,000	300,000	465	95.3	1,050,000	120,000	448	91.8
150,000	360,000	466	95.5	1,050,000	180,000	451	92.4
450,000	60,000	445	91.2	1,050,000	240,000	459	94.1
450,000	120,000	463	94.9	1,050,000	300,000	459	94.1
450,000	180,000	460	94.3	1,050,000	360,000	460	94.3
450,000	240,000	461	94.5	1,350,000	60,000	421	86.3
450,000	**300,000**	**467**	**95.7**	1,350,000	120,000	433	88.7
450,000	360,000	464	95.1	1,350,000	180,000	447	91.6
750,000	60,000	432	88.5	1,350,000	240,000	454	93.0
750,000	120,000	449	92.0	1,350,000	300,000	465	95.3
750,000	180,000	461	94.5	1,350,000	360,000	459	94.1

6.2 Three Comparisons of the Algorithms

In the experiments with PN^2, PDS, and PDS-PN all nodes evaluated during the search are counted; for the $\alpha\beta$ depth-first iterative-deepening searches nodes at depth i are counted only during iteration i. We adopted this method from Allis [1]. It makes a general comparison possible. The maximum number of nodes searched is 50,000,000. The limit corresponds roughly to tournament conditions. The maximum number of nodes stored in memory is 1,000,000. The parameters (a,b) of the growth function used in PN^2 are set at (1800K, 240K) according to the suggestions in Breuker *et al.* [8]. The parameter configuration (450K, 300K) found in the previous subsection will be used for PDS-PN. The smaller value of a corresponds to the smaller PN_1 trees resulting from the use of PDS-PN instead of PN^2. The fact that PDS is much slower than PN is an important factor too.

First Comparison

$\alpha\beta$, PN^2, PDS, and PDS-PN are tested on the same set of 488 forced-win LOA positions as described in Section 4. The results are given in Table 6. In the first column the four algorithms are mentioned. In the second column we see that 382 positions are solved[5] by $\alpha\beta$, 470 positions by PN^2, 473 positions by PDS, and 467 positions by PDS-PN. The set of 488 positions contains no position that only could be solved by $\alpha\beta$ search. In the third and fourth column the number of nodes and the time consumed are given for the subset of 371 positions, which all four algorithms are able to solve. A look at the third column shows that PDS search builds the smallest search trees and $\alpha\beta$ by far the largest. Like PN^2 and PDS, PDS-PN solves significantly more positions than $\alpha\beta$. This suggests that PDS-PN is a better endgame solver than $\alpha\beta$. As we have seen before, PN^2 and PDS-PN investigate more nodes than PDS, but both are still faster in CPU time than PDS for this subset. Due to the limit of 50,000,000 nodes and the somewhat lower search efficiency, PDS-PN solves three positions fewer than PN^2 (result PDS-PN is 99.4% with respect to PN^2) and six fewer than PDS (result PDS-PN is 98.7% with respect to PDS).

Table 6. Comparing the search algorithms on 488 test positions with a limit of 50,000,000 nodes

Algorithm	# of positions solved (out of 488)	accuracy (%)	371 positions	
			Total # of nodes	Total time (ms.)
$\alpha\beta$	382	78.3	2,645,022,391	33,878,642
PN^2	470	96.3	505,109,692	3,642,511
PDS	473	96.9	239,896,147	16,960,325
PDS-PN	467	95.7	924,924,336	5,860,908

Second Comparison

To investigate whether the memory restrictions are an actual obstacle we increased the limit of nodes searched to 500,000,000 nodes. In this second comparison PN^2 solves now 479 positions and PDS-PN becomes the best solver with a performance of 483 positions. The detailed results are given in Table 7.

The performance of PDS-PN in Table 7 is more effective than that of PN^2, viz. 483 to 479. However, we should thoughtfully take into account the condition for the total number of nodes searched and the time spent. Therefore, we continue our research in the direction of nodes searched and time spent with the 50,000,000 nodes limit. A reason for this decision is that the experimental time constraints are necessary for the PDS experiments.

[5] We remark that a slightly less inefficient version of our $\alpha\beta$ implementation could solve 383 positions (see Section 4).

Table 7. Comparing PN^2 and PDS-PN on 488 test positions with a limit of 500,000,000 nodes

Algorithm	# of positions solved (out of 488)	accuracy (%)	479 positions	
			Total # of nodes	Total time (ms.)
PN^2	479	98.2	2,261,482,395	13,295,688
PDS-PN	483	99.0	4,362,282,235	23,398,899

Third Comparison

For a better insight into the relation between PN^2, PDS, and PDS-PN we performed a third comparison. In Table 8 we provide the results of PN^2, PDS, and PDS-PN on a new subset of 457 positions of the principal test set, viz. all positions the three algorithms could solve under the 50,000,000 nodes limit condition. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the difference of performance is that for hard problems the PN_2-search tree becomes as large as the PN_1-search tree. Therefore, the PN_2-search tree is causing more overhead. However, if we look at the CPU time we see that PN^2 is almost 4 times faster than PDS. PDS has a relatively large time overhead because it performs multiple-iterative deepening at all nodes. PDS-PN searches 3.7 times more nodes than PDS but is still 3 times faster than PDS in CPU time. This is because PDS-PN is focussing more on the fast PN at the second level than on PDS at the first level. PDS-PN searches more nodes than PDS since the PN_2-search tree is repeatedly rebuilt and removed. The overhead is even bigger than PN^2's overhead because the children of the root of the PN_2-search tree are not stored (i.e., this is done to focus more on the fast PN search). It explains why PDS-PN searches 1.4 times more nodes than PN^2. Hence, our provisional conclusions are that on this set of 457 positions and under the 50,000,000 nodes condition: (1) PN^2 outperforms PDS-PN, and (2) PDS-PN is a faster solver than PDS and therefore more effective than PDS.

Table 8. Comparing PN^2, PDS and PDS-PN on 457 test positions (all solved) with a limit of 50,000,000 nodes

Algorithm	Total # of nodes	Total time (ms.)
PN^2	1,275,155,583	9,357,663
PDS	498,540,408	36,802,350
PDS-PN	1,845,371,831	11,952,086

6.3 Comparing the Algorithms for Hard Problems

Since the impact of the 50,000,000 nodes condition somewhat obscured our provisional conclusions above and since we felt that the 99.4% score by PDS-PN with respect to PN^2 was rather close, we performed a new experiment with a different set of LOA problems in an attempt to find more insights into the intricacies of these complex algorithms. In the new experiment PN^2 and PDS-PN are tested on a set of 286 LOA positions, which were on average harder than the ones in the previous test set.[6] In this context 'harder' means a longer distance to the final position (the solution), i.e., more time is needed. The conditions are the same as in the previous experiments except that the maximum number of nodes searched is set at 500,000,000. The PDS algorithm is not included because it takes too much time given the current node limit. In Table 9 we see that PN^2 solves 265 positions and PDS-PN 276. We remark that PN^2 solves 10 positions, which PDS-PN does not solve, but that PDS-PN solves 21 positions that PN^2 does not solve. The ratio in nodes and time between PN^2 and PDS-PN for the positions solved by both (255) is roughly similar to the previous experiments. The reason why PN^2 solves fewer positions than PDS-PN is its being restricted in working memory. We are in a delicate position since new experiments with much more working memory are now on the list to be performed. However, we assume that the nature of PN^2 with respect to using so much memory cannot be overcome. Hence we may conclude that within an acceptable time frame PDS-PN is a more effective endgame solver than PN^2 for hard problems.

Table 9. Comparing PN^2 and PDS-PN on 286 hard test positions with a limit of 500,000,000 nodes

Algorithm	# of positions solved (out of 286)	accuracy (%)	255 positions	
			Total # of nodes	Total time (ms.)
PN^2	265	92.7	10,061,461,685	57,343,198
PDS-PN	276	96.5	16,685,733,992	84,303,478

6.4 Comparing the Algorithms under Reduced Memory

From the experiments in the previous subsection it is clear that PN^2 will not be able to solve very hard problems since it will run out of working memory. To further solidify this statement experimentally, we tested the solving ability of PN^2 and PDS with restricted working memory. In these experiments we started with a memory capacity sufficient to store 1,000,000 nodes, subsequently we divided the memory capacity by two at each next step. The parameters a and b were also divided by two. The relation between memory and number of solved positions for both algorithms is given in Figure 7. We

[6] The test set can be found at www.cs.unimaas.nl/m.winands/loa/tscg2002b.zip.

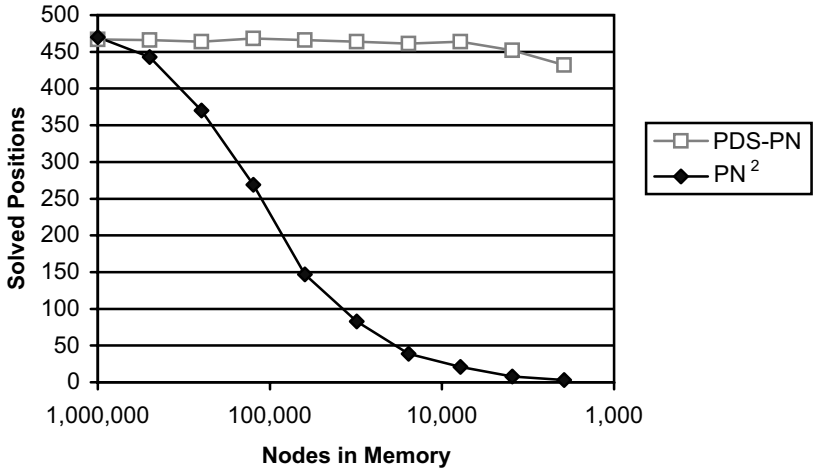


Fig. 7. Results with restricted memory

see that the solving performance rapidly decreases for PN^2. The performance of PDS-PN remains stable for a long time. Only when PDS-PN is restricted to fewer than 10,000 nodes, it begins to solve fewer positions. This experiment suggests that PDS-PN is to be preferred above PN^2 for the very hard problems when the memory capacity is in some way restricted. The reason is that PDS-PN is not suffering from memory constraints. If there are no memory constraints at all, PN^2 is preferred because under those circumstances it is the fastest algorithm.

7 Df-pn and PDS Comparison

Pawlewicz and Lew [24] used the set of 286 LOA positions of Subsection 6.3 to compare their implementation of df-pn and PDS with each other. Moreover, they enhanced df-pn and PDS with the $1 + \epsilon$ trick. To compare the speed differences in CPU time for each two methods, they calculated the geometric mean of the ratio of solving times (see Table 10). We see that the enhanced df-pn is the most efficient method, plain df-pn is the second best, and both PDS versions are the least efficient. Although the $1 + \epsilon$ trick was not designed for PDS, there is a noticeable difference between enhanced PDS and plain PDS in their experiments. For more details we refer to their paper [24].

Pawlewicz and Lew did neither implement nor tested PN^2 and PDS-PN on their hardware and with their data-structure implementation. Up to now there has not been a direct comparison between df-pn with PDS-PN using the same hardware and the same data-structure implementation. However, even so, we may still conclude that df-pn is an interesting alternative to PDS-PN. In Table 10 we see that df-pn is 4 times faster than PDS, while in Table 8 (Subsection 6.3) we see that PDS-PN is able to search 3 times faster than PDS.

Table 10. Comparison of df-pn and PDS on 286 hard test positions with a limit of 30 minutes by Pawlewicz and Lew [24]. The rate number r in row A and in column B indicates that algorithm A is r times faster than algorithm B

Algorithm	df-pn	df-pn $1+\varepsilon$	PDS	PDS $1+\varepsilon$
df-pn	1.00	0.63	2.83	2.64
df-pn $1+\varepsilon$	1.58	1.00	4.46	4.17
PDS	0.35	0.22	1.00	0.93
PDS $1+\varepsilon$	0.38	0.24	1.07	1.00

8 Conclusions and Future Research

Below we offer seven observations, three conclusions, and four suggestions for future research. Since observations and conclusions are intermingled, we present the conclusions in relation to the observations.

First, we have observed that mobility and deleting (dis)proved PN_2 subtrees speed up PN and PN^2 and increase their ability of solving endgame positions. Second, we have seen that the various PN-search algorithms outperform $\alpha\beta$ in solving endgame positions in LOA. Third, the memory problems make the plain PN search a weak solver for the harder problems. Fourth, PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$.

Our first conclusion is that PN and its variants offer a valuable tool for enhancing programs in endgames. We remark that PN^2 is still restricted by working memory, and that PDS is three times slower than PN^2 (Table 4) because of the delayed evaluation.

Our fifth observation is that PDS-PN is able to solve significantly more LOA endgame problems than $\alpha\beta$ search with enhancements. Our sixth observation is that the PDS-PN algorithm is almost as fast as PN^2 when the parameters for its growth function are chosen properly. It turns out that for each a it holds that the number of solved positions grows with increasing b, when the parameter b is still small. If b is sufficiently large, increasing it will not enlarge the number of solved positions. Our seventh observation states that (1) PDS-PN solves more hard positions than PN^2 within an acceptable time frame and (2) PDS-PN is more effective than PN or even PN^2 because it does not run out of memory for hard problems. Moreover, PDS-PN performs quite well under harsh memory conditions. This is especially appropriate for hard problems and for environments with very limited memory such as handheld computer platforms.

Hence, our second conclusion is that PDS-PN may be a more effective endgame solver for a set of hard problems than PDS and PN^2.

In this chapter we discussed the results of comparing PDS with df-pn as performed by Pawlewicz and Lew [24]. Df-pn was solving the set of hard problems 4 times faster than PDS. In Subsection 6.3 we saw that PDS-PN was able to search 3 times faster than PDS, whereas in Section 7 df-pn was

4 times faster than PDS. Although the conditions were different, our third conclusion is that df-pn may be an interesting alternative to PDS-PN.

Finally, we believe that there are four suggestions for the near future. The first challenge is testing PDS-PN in other domains with difficult endgames. An example of a game notoriously known for its complicated endgames is the game of Tsume-Shogi (a variant of Shogi). Several hard problems including solutions over a few hundred ply are solved by PN* [34] and PDS [20, 29]. It would be interesting to test PDS-PN on these problems. Second, it would be interesting to have a direct comparison between df-pn with PDS-PN using the same hardware and the same data-structure implementation. The third challenge would be to construct a two-level Proof-Number search variant using df-pn at the first level search, and a plain best-first PN search at the second level. Fourth, one problem clearly remains, viz. that there is no dynamic strategy available that determines when to use PN search instead of $\alpha\beta$ in a real game. This will be subject of future research as well.

Acknowledgements

The authors would like to thank the members of the MICC-IKAT Games and AI Group for their comments.

References

1. L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.
2. L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–123, 1994.
3. D.D. Berkey. *Calculus*. Saunders College Publishing, New York, NY, USA, 1988.
4. H.J. Berliner. The B*-tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.
5. D.M. Breuker. *Memory versus Search in Games*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.
6. D.M. Breuker, L.V. Allis, and H.J. van den Herik. How to mate: Applying proof-number search. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 251–272. University of Limburg, Maastricht, The Netherlands, 1994.
7. D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *ICCA Journal*, 19(3):175–180, 1996.
8. D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. The PN^2-search algorithm. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 115–132. IKAT, Universiteit Maastricht, Maastricht, The Netherlands, 2001.
9. D.M. Breuker, H.J. van den Herik, J.W.H.M. Uiterwijk, and L.V. Allis. A solution to the GHI problem for best-first search. *Theoretical Computer Science*, 252(1-2):121–149, 2001.

10. M. Campbell. The graph-history interaction: On ignoring position history. In *1985 Association for Computing Machinery Annual Conference*, page 278, 1985.
11. M. Campbell, A.J. Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
12. A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, 1998.
13. A. Kishimoto. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. PhD thesis, University of Alberta, Edmonton, Canada, 2005.
14. A. Kishimoto and M. Müller. Df-pn in Go: An application to the one-eye problem. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 125–141. Kluwer Academic Publishers, Boston, MA, USA, 2003.
15. A. Kishimoto and M. Müller. A solution to the GHI problem for depth-first proof-number search. *Information Sciences*, 175(4):296–314, 2005.
16. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
17. D.A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(1):278–310, 1988.
18. A. Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceedings of Complex Games Lab Workshop*, pages 40–45. ETL, Tsukuba, Japan, 1998.
19. A. Nagai. A new depth-first-search algorithm for AND/OR trees. Master's thesis, The University of Tokyo, Tokyo, Japan, 1999.
20. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and its Applications*. PhD thesis, The University of Tokyo, Tokyo, Japan, 2002.
21. A. Nagai and H. Imai. Application of df-pn+ to Othello endgames. In *Proceedings of Game Programming Workshop in Japan '99*, pages 16–23. Hakone, Japan, 1999.
22. E.V. Nalimov, G.M^{c}C. Haworth, and E.A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.
23. A.J. Palay. *Searching with Probabilities*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1983. Also published by Pitman, Boston, 1985.
24. J. Pawlewicz and Ł. Lew. Improving depth-first pn-search: $1 + \epsilon$ trick. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 160–171. Springer-Verlag, Heidelberg, Germany, 2007.
25. A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(2):255–293, 1996.
26. S. Sackson. *A Gamut of Games*. Random House, New York, NY, USA, 1969.
27. J-T. Saito, G. Chaslot, J.W.H.M. Uiterwijk, and H.J. van den Herik. Monte-carlo proof-number search for computer Go. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 50–61. Springer-Verlag, Heidelberg, Germany, 2007.
28. M. Sakuta. *Deterministic Solving of Problems with Uncertainty*. PhD thesis, Shizuoka University, Hamamatsu, Japan, 2001.
29. M. Sakuta and H. Iida. The performance of PN*, PDS and PN search on 6×6 Othello and Tsume-Shogi. In H.J. van den Herik and B. Monien, edi-

tors, *Advances in Computer Games 9*, pages 203–222. Universiteit Maastricht, Maastricht, The Netherlands, 2001.
30. J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.
31. J. Schaeffer. Game over: Black to play and draw in Checkers. *ICGA Journal*, 30(4):187–197, 2007.
32. J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.
33. J. Schaeffer and R. Lake. Solving the game of Checkers. In R.J. Nowakowski, editor, *Games of No Chance*, pages 119–133. Cambridge University Press, Cambridge, England, 1996.
34. M. Seo, H. Iida, and J.W.H.M. Uiterwijk. The PN*-search algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.
35. M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. PDS-PN: A new proof-number search algorithm: Application to Lines of Action. In J. Schaeffer, M. Müller, and Y. Björnsson, editors, *Computers and Games 2002*, volume 2883 of *Lecture Notes in Computer Science (LNCS)*, pages 170–185, Berlin, Germany, 2003. Springer-Verlag.
36. M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. An effective two-level proof-number search algorithm. *Theoretical Computer Science*, 313(3):511–525, 2004.
37. M.H.M. Winands, H.J. van den Herik, and J.W.H.M. Uiterwijk. An evaluation function for Lines of Action. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 249–260. Kluwer Academic Publishers, Boston, MA, USA, 2003.

Appendix: Pseudo Code for PN Search

Below we give the pseudo code for PN search, which was discussed in Subsection 2.1. For ease of comparison we use similar pseudo code as given in Breuker [5]. `PN(root, maxnodes)` is the main procedure of the algorithm. The procedure `evaluate(node)` evaluates a position, and assigns one of the following three values to a node: `FALSE`, `TRUE`, and `UNKNOWN`. The proof and disproof numbers of a node are initialised by `setProofAndDisproofNumbers(node)`. The function `selectMostProvingNode(node)` finds the most-proving node. Expanding the most-proving node is done by `expandNode(node)`. After the expansion of the most-proving node, the new information is backed up by `updateAncestors(node, root)`. The function `countNodes()` gives the number of nodes currently stored in memory.

```
//The PN-search algorithm
PN(root, maxnodes){

  evaluate(root);
  setProofAndDisproofNumbers(root);

  while(root.proof != 0 && root.disproof != 0
        && countNodes() <= maxnodes){
```

```
    //Second Part of the algorithm
    mostProvingNode = selectMostProvingNode(currentNode);
    expandNode(mostProvingNode);
    currentNode = updateAncestors(mostProvingNode, root);
  }
}

//Calculating proof and disproof numbers
setProofAndDisproofNumbers(node){
   if(node.expanded) //Internal node;
     if(node.type == AND_NODE){
       node.proof = 0;
       node.disproof = INFINITY;
       for(each child n){
         node.proof = node.proof + n.proof;
         if(n.disproof < node.disproof)
           node.disproof = n.disproof;
       }
     }
     else{ //OR node
       node.proof = ProofNode.INFINITY;
       node.disproof = 0;
       for(each child n){
         node.disproof = node.disproof + n.disproof;
         if(n.proof < node.proof)
           node.proof = n.proof;
       }
     }
   else //Leaf
     switch(node.value){
       case FALSE:
            node.proof = INFINITY;
            node.disproof = 0;
       case TRUE:
            node.proof = 0;
            node.disproof = INFINITY;
       case UNKNOWN:
            node.proof = 1;
            node.disproof = 1;
     }
}

//Select the most-proving node
SelectMostProvingNode(node){
  while(node.expanded){
    n = node.children;

    if(node.type == OR_NODE) //OR Node
      while(n.proof != node.proof)
```

```
          n = n.sibling;
       else //AND Node
         while(n.disproof != node.disproof)
           n = n.sibling;

        node = n;
     }
     return node;
  }

  //Expand node
  expandNode(node){
     generateAllChildren(node);
     for(each child n){
        evaluate(n);
        setProofAndDisproofNumbers(n);
        //Addition to original code
        if((node.type == OR_NODE && n.proof == 0) ||
           (node.type == AND_NODE && n.disproof == 0))
            break;
     }
     node.expanded = true;
  }

  //Update ancestors
  updateAncestors(node, root){

     do{
        oldProof = node.proof;
        oldDisProof = node.disproof;

        setProofAndDisproofNumbers(node);
        //No change on the path
        if(node.proof == oldProof &&
           node.disproof == oldDisProof)
          return node;
        //Delete (dis)proved trees
        if(node.proof == 0 || node.disproof == 0)
          node.deleteSubtree();

        if(node == root)
          return node;

        node = node.parent;
     }while(true)
  }
```

Index